PROJET IA

Problème du voyageur de commerce et algorithme génétique

Une image contenant texte, Police, Graphique, graphisme

Description générée automatiquement

- L3CILS -

**XIONG** Alexis : 20213544

**BOURDON** Laura : 20212795

Table des matières

[Problème Du Voyageur De Commerce 2](#_Toc164876986)

[Algorithme Génétique 3](#_Toc164876987)

[Algorithme Génétique & Problème Du Voyageur De Commerce 3](#_Toc164876988)

[Explication Des Méthodes 4](#_Toc164876989)

[Répartition Du Code 4](#_Toc164876990)

[Méthodes 5](#_Toc164876991)

[La méthode entryValid() 5](#_Toc164876992)

[La méthode afficherMatriceCarree(matrice) 5](#_Toc164876993)

[La méthode genererMatriceDistances(n) 5](#_Toc164876994)

[La méthode distance\_totale(circuit, matrice\_distance) 5](#_Toc164876995)

[La méthode générer\_population\_initiale(n, taille\_population) 6](#_Toc164876996)

[La méthode croisement(circuit1, circuit2) 6](#_Toc164876997)

[La méthode mutation(circuit) 6](#_Toc164876998)

[La méthode calculer\_fitness(circuit, matrice\_distances) 6](#_Toc164876999)

[La méthode selection\_par\_tri(population, fitness\_population) 6](#_Toc164877000)

[La méthode d'algorithme\_genetique(n, distance\_matrice, taille\_population, nb\_generation) 7](#_Toc164877001)

[La méthode circuits() 7](#_Toc164877002)

[La méthode graphe(circuit) 8](#_Toc164877003)

[La méthode affichage() 8](#_Toc164877004)

[Résultat 8](#_Toc164877005)

[VOIR ANNEXE 8](#_Toc164877006)

# Problème Du Voyageur De Commerce

Le voyageur de commerce souhaite passer par toutes les villes qu’il rencontre une seule fois de manière à être le plus rapide possible c’est-à-dire en prenant le chemin le plus court entre deux villes à chaque fois afin d’obtenir un circuit total le plus faible. Pour cela, on compare chacune des distances entre nos paires de villes ainsi que leur agencement général les unes par rapport aux autres.

# Algorithme Génétique

Un algorithme génétique, comme son nom l’indique, prend exemple sur ce qu’il se passe en génétique au niveau de la conservation et de la transmission ou non d’un gène présent dans une population. Au cours des générations, les gènes intéressants à la survie d’une espèce sont favorisés tandis que les gènes limitant les individus sont modifiés. Pour cela, des mutations ont lieu. Les mutations ne sont pas forcément désirées et ont une probabilité très faible de se produire. De plus, lors de la reproduction de deux d’entre eux, leur progéniture va récupérer leurs caractéristiques.

# Algorithme Génétique & Problème Du Voyageur De Commerce

L’algorithme génétique utilise les croisements et mutations pour la création de nouveaux individus, nous faisons de même avec notre population. Pour simuler une mutation, nous avons choisi d’interchanger deux villes d’un circuit de façon aléatoire. Une mutation a une probabilité très faible de se produire. Les croisements sont, quant à eux, effectués en prenant deux circuits distincts, chaque circuit est « coupé » en deux, la première partie du circuit1 et la seconde partie du circuit2 forment le nouvel individu1, la première partie du circuit2 et la seconde partie du circuit1 forment le nouvel individu2. Nous obtenons ainsi une nouvelle génération de dix individus, constituée des cinq meilleurs de notre première génération et des cinq nouveaux résultant des mutations et croisements. Nous répétons l’étape plusieurs fois afin d’avoir une population de plus en plus intéressante, c’est-à-dire des circuits avec des valeurs de plus en plus faibles. L’algorithme génétique nous permet donc de trouver le meilleur chemin pour notre voyageur.

# Explication Des Méthodes

## Répartition Du Code

Notre projet se découpe en plusieurs parties majeures : la création d’un premier échantillon de circuits, la mise en place de l’algorithme génétique et enfin l’affichage du graphe.

Nous commençons tout d’abord par demander à l’utilisateur le nombre de villes qu’il souhaite avoir. Nous vérifions que le texte entré est bel et bien un nombre compris entre trois et cinquante. Nous générons ensuite dix circuits aléatoires avec le nombre n de villes précédemment indiqué. Pour cela, nous sélectionnons le nom des n villes parmi notre liste de cinquante villes. Ces circuits constituent notre population initiale. Nous avons choisi d’afficher toutes les distances entre chaque couple de villes dans une matrice. La distance totale pour chaque circuit est également calculée.

Afin de créer notre prochaine génération, nous choisissons les cinq meilleurs circuits dans notre population grâce au fitness que nous avons défini. Cela nous permet d’avoir des parents intéressants et d’améliorer directement la nouvelle génération. Nous devons ensuite continuer de remplir cette nouvelle population. Pour cela, nous créons des enfants résultants du croisement entre deux des parents précédemment sectionnés. Nous appliquons par la suite une mutation à chacun des individus de notre nouvelle population. Notre population est ainsi créée. Nous créons plusieurs générations suivant le même principe. A chaque génération, le meilleur circuit est affiché et comparé avec le précédent. Si le meilleur circuit ne s’améliore plus, l’algorithme s’arrête et renvoie le meilleur circuit qu’il a trouvé, toujours grâce au fitness.

Au bout de la i-ème itération, nous obtenons notre population finale. Nous sélectionnons le circuit dont la distance est la plus faible c’est-à-dire le meilleur circuit et nous affichons le graphe qui en résulte, composé des n villes en tant que sommets et du chemin représenté par les arêtes.

L’utilisateur a la possibilité d’exécuter le code plusieurs fois d’affilée avec diverses valeurs grâce à la petite boîte affichée au lancement du code. Trois boutons sont disponibles : « Valider », « GO! » et « Afficher le graphe ». Ce premier bouton permet de confirmer l’entrée c’est-à-dire le nombre de villes que nous avons choisi et donc vérifier si les caractères tapés sont corrects. En revalidant ce bouton, l’utilisateur pourra de nouveau appuyer sur les boutons suivants et donc relancer le code autant de fois qu’il le souhaite. Le bouton « GO ! » lance l’exécution de l’algorithme génétique, c’est-à-dire la construction de toutes nos générations et la recherche du meilleur circuit possible dans notre population. Après avoir cliqué « GO ! », l’utilisateur peut afficher le graphe en cliquant sur « Afficher le graphe ». Il représente le meilleur circuit trouvé.

## Méthodes

### La méthode entryValid()

Cette méthode vérifie la validité des caractères entrés par l’utilisateur. Il doit avoir écrit des chiffres ou nombres compris entre trois et cinquante. Cette valeur n correspond au nombre de villes qui va constituer chaque individu de notre population. Si l’entrée n’est pas bonne, les boutons permettant d’accéder à la suite de l’algorithme ne seront pas disponibles. A chaque entrée valide, l’utilisateur peut relancer l’algorithme en cliquant sur les boutons « GO! » et « Afficher le graphe ». Cette méthode renvoie un booléen.

### La méthode afficherMatriceCarree(matrice)

Elle prend en paramètre la matrice des distances entre les villes. Son objectif est d'afficher la matrice afin de pouvoir observer plus simplement ce dernier. Pour cela, on récupère la taille n de la matrice puis calcule la longueur maximale de la matrice. Ensuite elle affiche chaque valeur de la matrice de sorte que la matrice soit carrée et que l'on puisse l'observer.

### La méthode genererMatriceDistances(n)

Elle prend en entrée le nombre de villes n afin de pouvoir créer un tableau à double dimension n\*n qui va pouvoir servir de matrice. Au sein de cette matrice, nous allons ajouter une valeur entière aléatoire entre 1 à 100 qui vont nous simuler la distance entre les différentes villes. De plus, pour le réalisme des distances entre les villes, la matrice sera symétrique avec une diagonale nulle. La diagonale nulle représente une distance entre une ville et elle-même. Par conséquent, la valeur sera nulle et les distances entre la ville A vers la ville B ainsi la distance entre la ville B vers la ville A seront égales.

### La méthode distance\_totale(circuit, matrice\_distance)

Elle prend en entrée un circuit, qui représente un tableau de villes ainsi qu'une matrice des distances entre les villes. Nous commençons par initialiser la valeur de distance\_totale à 0, puis dans une boucle parcourant la longueur du circuit, nous allons additionner les villes deux par deux jusqu'à avoir la distance totale. La matrice nommée matrice\_distances nous servira à aller récupérer la valeur de la distance entre les villes sélectionnées.

### La méthode générer\_population\_initiale(n, taille\_population)

Comme son nom l’indique, cette méthode permet de générer notre toute première population. Elle prend en argument le nombre n de villes et le nombre d’individus constituant notre population que nous avons défini à dix individus. Elle choisit les noms des n villes aléatoirement parmi notre liste de villes et n’utilise un nom qu’une seule et unique fois pour éviter les doublons. Cette méthode renvoie une liste.

### La méthode croisement(circuit1, circuit2)

Cette méthode permet de créer deux nouveaux circuits enfants résultants des deux circuits parents pris en argument. Pour cela, on choisit au hasard où couper le premier parent, on ajoute les villes contenues dans cette première partie, puis on complète avec les villes du deuxième parent qui ne sont pas encore dans notre enfant, on génère le deuxième enfant de la même manière mais en échangeant l’ordre des deux parents. La méthode renvoie donc deux nouveaux circuits.

### La méthode mutation(circuit)

Pour effectuer une mutation, nous avons choisi de sélectionner au hasard deux index de notre circuit et d’échanger de place les villes situées à cet endroit. Le taux de mutation est choisi faible. La méthode mutation(circuit) renvoie le circuit qui a été muté.

### La méthode calculer\_fitness(circuit, matrice\_distances)

Cette méthode permet d’évaluer un circuit. Pour cela, on divise un par la distance totale de notre circuit. Cette distance est calculée à partir de la matrice prise en entrée. Plus la valeur obtenue est grande, plus la distance est petite et donc plus le circuit est intéressant pour notre problème. On renvoie la valeur du fitness en sortie.

### La méthode selection\_par\_tri(population, fitness\_population)

Elle prend en entrée une population avec son fitness. Nous avons créé une variable population\_triee qui a pour but de trier la population dans l'ordre croissant afin d'obtenir les distances totales les plus courts en début de liste. Ensuite, nous utilisons une variable nommée parents qui prend les 4 premiers éléments de population\_triee et la méthode finit par retourner les parents. Cette méthode va nous permettre de selectionner les 4 individus ayant les plus courtes distances afin de pouvoir l'utiliser dans l'algorithme génétique.

### La méthode d'algorithme\_genetique(n, distance\_matrice, taille\_population, nb\_generation)

Cette méthode prend en valeur un nombre de ville n saisissable dans l'interface graphique, la matrice des distances entre les villes, la taille de la population et le nombre de génération. Cette méthode va appliquer le principe de l'algorithme génétique en commençant par créer la population initiale avec la méthode generer\_population\_initiale(), elle va ensuite conserver le fitness de la population dans un tableau et garder le meilleur circuit initial. Ensuite, pour chaque génération, nous allons créer une nouvelle population en créant un tableau initialement vide nommée nouvelle\_population et selectionner les 4 parents avec la méthode selection\_par\_tri(). Par la suite, nous allons insérer les parents au sein de la nouvelle population tout en les mutant en utilisant la méthode mutation() et tant que la taille de la nouvelle population n'est pas la même que celle de l'ancienne, nous allons créer deux nouveaux enfants à partir de 2 parents parmi les 4 et utilisons la méthode croisement() pour mélanger les gènes des 2 parents, nous appliquons une mutation à ses enfants et enfin nous les ajoutons à la nouvelle population. La nouvelle population remplace maintenant l'ancienne et nous comparons a nouveau leur fitness en utilisant la méthode calculer\_fitness() pour avoir toutes leurs fitness et en mettant dans la variable nouveau\_meilleur\_circuit l'individu qui a la distance totale la plus faible et la comparons avec meilleur\_circuit qui correspond au meilleur circuit de l'ancienne génération. S'il est différent de l'ancien meilleur circuit alors on le remplace et on continue, sinon on incrémente la variable meilleures\_generations\_sans\_amelioration. Dans notre algorithme génétique, il existe 2 conditions d'arrêts, le nombre de génération qui est l'équivalent de 3 fois le nombre de ville afin d'avoir un temps de calcul plus ou moins long en fonction du nombre de villes, ainsi qu'un compteur de nombre de génération sans amélioration initialisé à 5. En effet, si le meilleur circuit est toujours identique cela signifie que l'algorithme n'arrive plus à trouver de meilleurs circuits.

### La méthode circuits()

circuits() permet d’appeler les méthodes et de déclarer les variables nécessaires à l’exécution de la méthode algorithme\_genetique(n, distance\_matrice, taille\_population, nb\_generation). Cette méthode permet également d’afficher notre matrice de distances.

### La méthode graphe(circuit)

On affiche le graphe représentant notre meilleur circuit au bout de la i-ème génération grâce à cette méthode. Chaque ville correspond à un sommet du graphe. L’orientation des arêtes représente l’ordre de visite des villes que le voyageur de commerce devra effectuer.

### La méthode affichage()

Cette méthode permet simplement de faciliter l’utilisation des boutons et d’appeler la méthode graphe(circuit).

# Résultat

L’algorithme génétique ne permet pas toujours d’obtenir le circuit le plus court rencontré lors des diverses générations mais un des circuits les plus courts. En effet, l’application des mutations et des croisements sur notre population peut parfois avoir l’effet inverse de celui escompté et rendre la génération suivante moins bonne que la précédente. Il est aussi possible, bien que très peu probable, que notre dernière génération soit loin de la meilleure en termes de distance.

# VOIR ANNEXE